

"Express Mail" mailing label number:

EL675711178US

## LOCK FREE REFERENCE COUNTING

David L. Detlefs,  
Paul A. Martin  
Mark S. Moir, and  
Guy L. Steele Jr.

### CROSS-REFERENCE TO RELATED APPLICATION(S)

[1001] This application claims benefit of U.S. Provisional Application No. 60/261,633, filed January 12, 2001.

Sub A  
[1002] In addition, this application is related to (1) U.S. Patent Application No. <not yet assigned> (docket 004-4666-1), entitled "LOCK-FREE IMPLEMENTATION OF CONCURRENT SHARED OBJECT WITH DYNAMIC NODE ALLOCATION AND DISTINGUISHING POINTER VALUE," naming Guy L. Steele Jr., Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, Mark S. Moir and David L. Detlefs as inventors, and filed on even date herewith and (2) U.S. Patent Application No. <not yet assigned> (docket 004-4665-1), entitled "CONCURRENT SHARED OBJECT IMPLEMENTED USING A LINKED-LIST WITH AMORTIZED NODE ALLOCATION," naming Paul A. Martin, David L. Detlefs, Alexander T. Garthwaite, Guy L. Steele Jr. and Mark S. Moir as inventors, and filed on even date herewith.

### BACKGROUND OF THE INVENTION

#### Field of the Invention

[1003] The present invention relates generally to coordination amongst execution sequences in a multiprocessor computer, and more particularly, to structures and techniques for facilitating non-blocking access to concurrent shared objects.

#### Description of the Related Art

[1004] It is well known that availability of automatic dynamic memory management facilities, often referred to as garbage collection (GC), can simplify the design of

sequential implementations of data structures. Furthermore, availability of garbage collection facilities can simplify the handling of various synchronization issues in the design of concurrent data structures.

[1005] For example, concurrent data structure implementations for an execution environment that provides garbage collection can benefit from a free solution to the so-called ABA problem. The ABA problem arises when a data system fails to detect that a value changes (e.g., from A to B) and then changes back to its original value (i.e., A). For example, if a compare-and-swap-type operation (e.g., a CAS or DCAS operation) is about to operate on a pointer and the object to which it points is freed and then reallocated, then it is possible for the CAS or DCAS to succeed even though it should fail. In a proper implementation, this possibility should be prevented by ensuring that an object is not freed while an execution thread has pointers to it. Garbage collector implementations typically ensure such behavior, e.g., by stopping a thread and inspecting its registers and stack for pointers. However, in the absence of a garbage collector, responsibility for handling the ABA problem generally falls to the concurrent data structure implementation itself.

[1006] While use of a garbage-collected execution environment is possible in some situations, such an environment is not always available. For example, many common programming and execution environments do not support garbage collection. Second, even those environments that do support garbage collection typically employ excessive synchronization, such as locking and/or stop-the-world mechanisms, which impair scalability and may be impractical for certain applications. Finally, concurrent shared object implementations that depend on the existence of a garbage collection facility are generally not suitable for use in the implementation of the garbage collector itself.

[1007] In view of the above, structures and techniques are desired that would allow programmers to exploit the advantages of a garbage collected execution environment in the design of data structure implementations despite the absence of a garbage collection facility.

## **SUMMARY OF THE INVENTION**

[1008] Accordingly, we have developed a methodology for transforming garbage collection-dependent algorithms and/or shared object implementations into a form that no longer presumes the existence of an independent, or execution environment provided, garbage collector. Transformed algorithms and shared object implementations provide explicit reclamation of storage using lock-free pointer operations. The transformations have been applied to lock-free algorithms and shared object implementations and preserve lock-freedom of such algorithms and implementations. As a result, existing and future lock-free algorithms and shared object implementations that depend on a garbage-collected execution environment can be exploited in environments that do not provide garbage collection. Furthermore, algorithms and shared object implementations that employ explicit reclamation of storage using lock-free pointer operations such as described herein may be employed in the implementation of a garbage collector itself.

[1009] While some realizations of the present invention include transformation techniques or facilities, others realizations include the lock-free implementations themselves (whether or not derived through such transformations). For example, some realizations include a shared object or data structure that employs lock-free pointer operations. Some realizations include multiprocessor systems and/or computer program products incorporating explicit reclamation of storage using lock-free pointer operations. In some multiprocessor environments, algorithms and/or shared object implementations employed by operating system or execution environment facilities (including, without limitation, garbage collection or other automatic dynamic memory management facilities) may exploit the techniques described herein to explicitly reclaim storage.

[1010] In one embodiment in accordance with the present invention, a method of providing storage reclamation in a multiprocessor computer system includes maintaining respective reference counts for shared objects, accessing pointers to the shared objects using lock-free pointer operations to coordinate modification of respective reference counts and freeing storage associated with a particular one of the shared objects only once the corresponding reference count indicates that the

particular shared object is unreferenced. In some realizations, the lock-free pointer operations ensure that if a number of pointers referencing the particular shared object is non-zero, then so too is the corresponding reference count and that if no pointers reference the particular shared object, then the corresponding reference count eventually becomes zero. DCAS primitives, CAS primitives or other synchronization primitives are employed in some realizations of the lock-free pointer operations. Of course, the assertion (here and elsewhere) that the reference count eventually becomes zero assumes that no processes fail. If a process does fail, it can prevent storage from being reclaimed by failing to decrement a reference count to zero. Accordingly, some realizations may employ additional techniques such a second-tier garbage-collector or a specialized process failure handler to reclaim such storage.

[1011] In another embodiment in accordance with the present invention, a lock-free implementation of a concurrent shared object includes plural component shared objects encoded in dynamically-allocated shared storage and access operations. Prior to attempting creation or replication of a pointer to any of the component shared objects, the access operations increment a corresponding reference count, and upon failure of the attempt, thereafter decrement the corresponding reference count. Except when handling a pointer creation failure, the access operations decrement a particular reference count no earlier than upon destruction of a pointer to a corresponding one of the component shared objects. In some realizations, the access operations employ lock-free, reference-count-maintaining pointer operations, such as a lock-free, reference-count-maintaining load, a lock-free, reference-count-maintaining store, a lock-free, reference-count-maintaining copy, a lock-free, reference-count-maintaining destroy, a lock-free, reference-count-maintaining compare-and-swap (CAS) and/or a lock-free, reference-count-maintaining double compare-and-swap (DCAS).

[1012] In still another embodiment in accordance with the present invention, a method of transforming an implementation of a concurrent shared data structure from garbage collection- (GC-) dependent to GC-independent form includes associating a reference count with each shared object instance, modifying the implementation, if necessary, to ensure cycle-free garbage, replacing pointer accesses in the implementation with corresponding lock-free, reference-count-maintaining counterpart operations, and explicitly managing local pointer variables using a lock-

free, reference-count-maintaining destroy operation that frees storage if a corresponding reference count has reached zero. In some realizations, the replacement of pointer accesses includes replacing an access that assigns a shared pointer value to a local pointer variable with a lock-free, reference-count-maintaining load operation, replacing an access that assigns a local pointer value to a shared pointer variable with a lock-free, reference-count-maintaining store operation, and/or replacing an access that assigns a local pointer value to a local pointer variable with a lock-free, reference-count-maintaining copy operation.

[1013] In some realizations, the method further includes generating a computer program product including a computer readable encoding of the concurrent shared data structure, which is instantiable in dynamically-allocated shared storage, the computer readable encoding further including functional sequences that facilitate access to the concurrent shared data structure and that include the lock-free, reference-count-maintaining counterpart operations.

[1014] In still yet another embodiment in accordance with the present invention, a computer program product includes a representation of a shared object that is instantiable as zero or more component objects in dynamically-allocated shared storage of a multiprocessor and at least one instruction sequence executable by respective processors of the multiprocessor. The instruction sequence implements at least one access operation on the shared object and employs one or more lock-free pointer operations to maintain reference counts for one or more accessed component objects thereof. The instruction sequence further implements explicit reclamation of the component objects, thereby freeing storage associated with a particular one of the component objects only once the corresponding reference count indicates that the particular component object is unreferenced. In some realizations, the component objects of the shared object are organized as a linked-list and the access operation supports concurrent access to the linked-list. In some variations, the computer program product at least partially implements a mutator that provides explicit reclamation of the dynamically-allocated shared storage. In some variations, the computer program product at least partially implements a garbage collector that reclaims shared storage dynamically-allocated for a mutator and, which employs the shared object in coordination thereof.

[1015] In still yet another embodiment in accordance with the present invention, an apparatus includes multiple processors, one or more stores addressable by the processors, one or more shared pointer variables accessible by each of the processors for referencing a shared object encoded in the stores, and means for coordinating competing access to the shared object. The means for coordinating competing access using one or more reference counts and pointer manipulations that employ one or more lock-free pointer operations to ensure that if the number of pointers the shared object is non-zero, then so too is the corresponding reference count and further that if no pointers reference the shared object, then the corresponding reference count eventually becomes zero. In some realizations, the apparatus further includes means for freeing the shared object only once the corresponding reference count indicates that the shared object is unreferenced.

#### **BRIEF DESCRIPTION OF THE DRAWINGS**

[1016] The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings.

[1017] **FIGS. 1A and 1B** illustrate before and after states of a simple data structure in which reference counts are provided.

[1018] **FIG. 2** depicts a shared memory multiprocessor configuration that serves as a useful illustrative environment for describing operation of some shared object implementations in accordance with the present invention.

[1019] **FIGS. 3A and 4A and FIGS. 3B and 4B** illustrate corresponding aspects of an exemplary concurrent shared object implementation and a GC-independent implementation derived therefrom. **FIGS. 3A and 4A and FIGS. 3B and 4B** are presented in side-by-side fashion to illustrate the correspondence.

[1020] The use of the same reference symbols in different drawings indicates similar or identical items.

**DESCRIPTION OF THE PREFERRED EMBODIMENT(S)**

[1021] Our techniques are based on reference counting, which persons of ordinary skill in the art will recognize as one of the classical methods of storage reclamation. *See generally*, Jones and Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons Ltd. (1996), pp. 19-25 (reviewing reference counting). However, our techniques use a synchronization primitive, such as a double compare and swap (DCAS) operation, to update the reference count of an object atomically with a check that a pointer to the object still exists. By weakening the typical reference counting implementation invariant that reference counts record the exact number of pointers to an object, we are able to separate the updates of reference counts from the updates of the pointers themselves. This allows us to support strong synchronization operations, including CAS and DCAS, on pointers. In contrast, classical reference counting implementations for concurrent environments typically require the use of locks on all objects that might be shared between threads. *See generally*, Jones and Lins, pp. 200-202 (reviewing concurrent reference counting methods).

[1022] We have designed our methodology to preserve lock-freedom. That is, if a GC-dependent implementation is lock-free, then so too will be a GC-independent implementation derived using our methodology. In this regard, the assertion of lock-freedom may ignore certain operating system or execution environment provided facilities such as for creation and destruction of objects. For example, in many systems, `malloc` and `free` (or other similar facilities) are provided. Unfortunately, implementations of such facilities are often not lock-free, and therefore, strictly-speaking, implementations based thereon are not themselves lock-free. However, most production-quality `malloc/free` implementations do attempt to avoid contention for locks—for example, by maintaining separate allocation buffers for each thread—and therefore avoid most of the problems associated with locks. As a result, while some exploitations of the present invention build upon underlying facilities, which are truly lock-free, others need not. Accordingly, even concurrent shared object implementations for execution environments in which implementations of `malloc`, `free` or similar facilities are not, strictly speaking, lock-free derive substantial benefits from our techniques.

[1023] As used herein, an implementation of a concurrent data structure is said to be *lock-free* if it guarantees that in any execution, after a finite number of steps of one of its operations, some operation on the data structure completes. Of course, it is possible that the operating system, execution environment or garbage collector might prevent threads from executing any instructions, in which case, no operations will be completed on the data structure. However, this does not mean that the concurrent data structure implementation is not lock-free. Thus, it is not contradictory to talk about a lock-free GC-dependent concurrent data structure implementation, even in environments in which a garbage collector, operating system or execution environment facility is blocking. Lock-free programming is increasingly important for overcoming the problems associated with locking, including performance bottlenecks, susceptibility to delays and failures, design complications, and, in real-time systems, priority inversion.

[1024] Our methodology is based on the well-known garbage collection technique of *reference counting*. We refer to our methodology as LFRC (Lock-Free Reference Counting). In some realizations, each object includes or has an associated count of the number of pointers that reference it. Such objects are freed if and only if this count reaches zero. Persons of ordinary skill in the art will recognize that one reason that typical tracing-type garbage collectors “stop the world” (e.g., by suspending all mutator threads) is that some of these pointers can reside in registers and/or stacks and that discovery of these pointers may require operating system support. As a result, garbage collection is difficult to perform concurrently with execution of mutator threads. Accordingly, one of our goals is to enable programmers of multiprocessor systems to take advantage of the simplicity afforded by the existence of automatic dynamic memory management, without having to use locks or stop-the-world techniques.

[1025] As an initial proposition, to maintain accurate reference counts, we would like to atomically create a pointer to an object and increment that object's reference count, and to atomically destroy a pointer to an object and decrement its reference count.

**FIGS. 1A and 1B** illustrate appropriate reference count updates when a pointer value in field 111 of object 101 is changed to reference object 103 rather than object 102.

**FIG. 1A** illustrates the pre-change states of object 101, 102 and 103, while **FIG. 1B**



illustrates the post-change states. Reference count fields 112 and 113 of the respective objects are decremented and incremented accordingly. In this way, each object's reference count always accurately reflects the number of pointers to it. By freeing an object when and only when its reference count becomes zero, it is possible to ensure that objects are not freed prematurely, but that they are eventually freed when no pointers to the object remain.

[1026] The main difficulty that arises in the above-described approach is the need to atomically modify three separate memory locations, e.g., the pointer store and the reference count fields of the objects to which the store points both before and after update. This goal can be achieved using synchronization primitives to enforce atomicity. For example, in some realizations we assume the availability of compare-and-swap (CAS) and double compare-and-swap (DCAS) operations. Other realizations may employ other synchronization primitives or facilities such as those provided by transactional memory.

[1027] CAS operations atomically access a single memory location, while DCAS operations atomically access two independently-chosen memory locations. DCAS operations are well known in the art and have been implemented in hardware, such as in the Motorola 68040 processor, as well as through software emulation. Accordingly, a variety of suitable implementations exist and the descriptive code that follows is meant to facilitate later description of concurrent shared object implementations in accordance with the present invention and not to limit the set of suitable DCAS implementations. For example, order of operations is merely illustrative and any implementation with substantially equivalent semantics is also suitable.

```
boolean DCAS(val *addr1, val *addr2,
             val old1, val old2,
             val new1, val new2) {
    atomically {
        if ((*addr1==old1) && (*addr2==old2)) {
            *addr1 = new1;
            *addr2 = new2;
            return true;
        } else {
            return false;
        }
    }
}
```

}  
}

[1028] The sequence of operations implementing the DCAS operation is executed atomically using support suitable to the particular realization. Persons of ordinary skill in the art will appreciate a variety of suitable implementations. For example, in some realizations, atomicity is provided using hardware support (e.g., as implemented by the Motorola 68040 microprocessor or as described in M. Herlihy and J. Moss, *Transactional memory: Architectural Support For Lock-Free Data Structures*, Technical Report CRL 92/07, Digital Equipment Corporation, Cambridge Research Lab, 1992, 12 pages) or using non-blocking software emulation (such as described in G. Barnes, *A Method For Implementing Lock-Free Shared Data Structures*, in *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, June 1993 or in N. Shavit and D. Touitou, *Software transactional memory*, *Distributed Computing*, 10(2):99–116, February 1997). Although the above-referenced implementation is presently preferred, other DCAS implementations that substantially preserve the semantics of the descriptive code (above) are also suitable.

[1029] Unfortunately, even a DCAS operation is not strong enough to allow us to maintain reference counts that are accurate at all times. For example, referring to the illustration of **FIGS. 1A and 1B**, if a pointer value in shared memory (e.g., that stored in field 111 of object 101) points to object 102, and we change the pointer value to instead point to another object 103, then we would have to atomically (i) modify the pointer (i.e., the contents of field 111), (ii) increment reference count 113 of object 103 and (iii) decrement reference count 112 of object 102. However, it turns out that a weaker requirement on the reference counts suffices, and that this requirement *can* be achieved using a DCAS operation. This weakening is based on the insight that reference counts do not always need to be accurate. Instead, it is sufficient that if the number of pointers to an object is non-zero, then so too is its reference count, and that if the number of pointers is zero, then the reference count eventually becomes zero. These two requirements respectively guarantee that an object is never freed prematurely, and that the reference count of an object that has no pointers to it eventually become zero, so that it can be freed.

[1030] Furthermore, based on the above insight, it is safe for a thread to increment an object's reference count before creating a new pointer to it, provided that the thread eventually either creates the pointer or, if not, decrements the reference count to compensate for the previous increment. It is tempting to think that the weaker requirements can be achieved without resorting to use DCAS. However, when we load a pointer from a shared memory location, we need to increment the reference count of the object to which the loaded value points. If we can access this reference count only with a single-variable CAS operation, then there is a risk that the object will be freed before we increment the reference count, and that the subsequent attempt to increment the reference count will corrupt memory that has been freed, and potentially reallocated for another purpose. By using a DCAS operation, we can increment the reference count while atomically ensuring that a pointer to the object still exists.

### **Lock-Free Reference Counting**

[1031] Some lock-free reference counting (LFRC) implementations in accordance with the present invention provide a set of pointer operations that mediate access and manipulation of pointers consistent with the goals described above. For example, an LFRC implementation may provide a set of operations that support loading, storing, copying, and destroying pointers. For many of the concurrent shared object implementations of interest, additional operations may be provided. For example, to support concurrent shared object implementations in which compare-and-swap (CAS) and double compare-and-swap (DCAS) constructs are employed to mediate concurrent access to the shared object, LFRC pointer operations that provide CAS and DCAS semantics may also be provided.

[1032] The description that follows focuses on implementations, techniques and methodologies that build upon an illustrative set of pointer operations by which load, store, copy, destroy, CAS and DCAS operations on pointers are supported. However, the particular set, while generally sufficient to support a wide range of concurrent data structure implementations, is merely illustrative. Based on the description herein, persons of ordinary skill in the art will appreciate implementations of suitable pointer operation sets in which access and manipulation of pointers is performed consistent

with the goals described above. For example, some useful additional pointer operations include operations to facilitate passing pointers by value and operations that combine allocation and pointer initialization functions. Suitable pointer operation sets may include LFRC versions of other synchronization primitives, such as the load-linked (LL) and store-conditional (SC) synchronization primitives.

[1033] We present a methodology for transforming any garbage-collection-dependent concurrent data structure implementation (lock-free or otherwise) that satisfies the two criteria below into an equivalent implementation that does not depend on garbage collection.

1. **LFRC Compliance** The implementation does not access or manipulate pointers other than through a set of pointer operations that ensure that if the number of pointers to an object is non-zero, then so too is its reference count, and that if the number of pointers is zero, then the reference count eventually becomes zero. For example, compliance with such a criterion generally precludes the use of pointer arithmetic, unless the implementation thereof enforces the criterion. For example, in some implementations, arithmetic operations on pointers could be overloaded with compliant versions of the arithmetic operations. In an illustrative realization described below, an implementation of a concurrent shared object accesses and manipulates pointers only through a set of functions, procedures or methods (e.g., load, store, copy, destroy, CAS and/or DCAS operations) that ensure compliance.
2. **Cycle-Free Garbage** There are no pointer cycles in garbage. Note that, cycles may exist in the concurrent data structure, but not amongst objects that have been removed from the data structure, and which should be freed.

[1034] The transformation preserves lock-freedom. In particular, if the original implementation is lock-free, so too is the garbage-collection-independent algorithm produced by applying our methodology.

LFRC Operations – An Illustrative Set

[1035] Subject to the foregoing, an illustrative set of LFRC pointer operations is now described. As stated above, we assume that pointers in a data structure implementation under consideration are accessed *only* by means of these operations.

1. LFRCLoad (A, p) — A is a pointer to a shared memory location that contains a pointer, and p is a pointer to a local pointer variable. The effect is to load the value from the location pointed to by A into the variable pointed to by p.
2. LFRCLoad (A, v) — A is a pointer to a shared memory location that contains a pointer, and v is a pointer value to be stored in this location.
3. LFRCCopy (p, v) — p is a pointer to a local pointer variable and v is a pointer value to be copied to the variable pointed to by p.
4. LFRCDestroy (v) — v is the value of a local pointer variable that is about to be destroyed.
5. LFRCCAS (A0, old0, new0) — A0 is a pointer to a shared memory location that contains a pointer, and old0 and new0 are pointer values. The effect is to atomically compare the contents of the location pointed to by A0 with old0 and to change these contents to new0 and return *true* if the comparison succeeds; if it fails, then the contents of the location pointed to by A0 are left unchanged, and LFRCCAS returns *false*.
6. LFRCDCAS (A0, A1, old0, old1, new0, new1) — A0 and A1 are pointers to shared memory locations that contain pointers, and old0, old1, new0, and new1 are pointer values. The effect is to atomically compare the contents of the location pointed to by A0 with old0 and the contents of the location pointed to by A1 with old1, to change the contents of the locations pointed to by A0 and A1 to new0 and new1, respectively, and to return *true* if the comparisons both succeed; if either comparison fails, then the contents of the locations pointed to by A0 and A1 are left unchanged, and LFRCDCAS returns *false*.

[1036] **FIG. 2** depicts a shared memory multiprocessor configuration in which the illustrated set of LFRC pointer operations may be employed. In particular, **FIG. 2**

depicts a pair of processors **211** and **212** that access storage **240**. Storage **240** includes a shared storage portion **230** and local storage portions **221** and **222**, respectively accessible by execution threads executing on processors **211** and **212**. In general, the multiprocessor configuration is illustrative of a wide variety of physical implementations, including implementations in which the illustrated shared and local storage portions correspond to one or more underlying physical structures (e.g., memory, register or other storage), which may be shared, distributed or partially shared and partially distributed.

[1037] Accordingly, the illustration of **FIG. 2** is meant to exemplify an architectural view of a multiprocessor configuration from the perspective of execution threads, rather than any particular physical implementation. Indeed, in some realizations, data structures encoded in shared storage portion **230** (or portions thereof) and local storage (e.g., portion **221** and/or **222**) may reside in or on the same physical structures. Similarly, shared storage portion **230** need not correspond to a single physical structure. Instead, shared storage portion **230** may correspond to a collection of sub-portions each associated with a processor, wherein the multiprocessor configuration provides communication mechanisms (e.g., message passing facilities, bus protocols, etc.) to architecturally present the collection of sub-portions as shared storage. Furthermore, local storage portions **221** and **222** may correspond to one or more underlying physical structures including addressable memory, register, stack or other storage that are architecturally presented as local to a corresponding processor. Persons of ordinary skill in the art will appreciate a wide variety of suitable physical implementations whereby an architectural abstraction of shared memory is provided. Realizations in accordance with the present invention may employ any such suitable physical implementation.

[1038] In view of the foregoing and without limitation on the range of underlying physical implementation of the shared memory abstraction, LFRC pointer operations may be better understood as follows. Pointer A references a shared memory location **231** that contains a pointer to an object **232** in shared memory. One or more pointers such as pointer A is (are) employed as operands of the LFRCLoad, LFRStore, LFRCCAS and LFRCDCAS operations described herein. Similarly, pointer p references local storage **234** that contains a pointer to an object (e.g., object **232**) in shared

memory. In this regard, **FIG. 2** illustrates a state,  $*A == *p$ , consistent with successful completion of either a LFRCLoad or LFRCLoad operation. In general, pointers A and p may reside in any of a variety storage locations. Often, both pointers reside in storage local to a particular processor. However, either or both of the pointers may reside elsewhere, such as in shared storage.

[1039] In our experience, the operations presented here are sufficient for many concurrent shared object implementations, but can result in somewhat non-transparent code. Accordingly, we have also implemented some extensions that allow more elegant programming and handle issues such as the pointer created by passing a pointer by value transparently. For example,

1.  $p = \text{LFRCLoad2}(A)$  — A is a pointer to a shared memory location that contains a pointer, and p is a local pointer variable, where p is known not to contain a pointer (e.g., it has just been declared). The effect is to load the value from the location pointed to by A into p.
2.  $\text{LFRCLoadAlloc}(A, v)$  — A is a pointer to a shared memory location that contains a pointer, and v is a pointer value that will not be used (or destroyed) again. Accordingly, there is no need to increment a reference count corresponding to v. This variation is useful when we want to invoke an allocation routine directly as the second parameter, e.g., as  $\text{LFRCLoadAlloc}(\&X, \text{allocate\_structure}())$ .
3.  $\text{LFRCLDCAS2}(A0, A1, \text{old0}, \text{old1}, \text{new0}, \text{new1})$  — A0 is a pointer to a shared memory location that contains a pointer, A1 is a pointer to a shared memory location that contains a non-pointer value, old0 and new0 are pointer values, and old1 and new1 are values, e.g., literals, for which no reference counting is required.
4.  $\text{LFRCLPass}(p)$  — p is a pointer value to be passed by value and for which a reference count should be incremented. This variation is useful when we want to pass p to a routine, e.g., as  $\text{Example}(\dots, \text{LFRCLPass}(p))$ .

[1040] Based on the description herein, persons of ordinary skill in the art will appreciate variations of the described implementations, which may employ these and other extensions and/or variations on a set of supported pointer operations.

### **LFRC Methodology**

[1041] Building on the previously described illustrative set of pointer operations, we describe now steps of a methodology for transforming a GC-dependent implementation into a GC-independent implementation. In general, these steps (with the possible exception of step 4) can be automated using facilities of an object-oriented language such as C++. Based on the description herein, persons of ordinary skill in the art will appreciate a variety of facilities that could be employed. *See e.g., Detlefs, Garbage Collection and Run-Time Typing as a C++ Library, in Proceedings of the 1992 Usenix C++ Conference, pp. 37-56, 1992* (describing a suitable "smart pointer" technique). Nonetheless, presenting these steps explicitly makes our presentation clearer, and serves as a roadmap for exploitation of our methodology in non-object-oriented languages as well. Accordingly, one suitable transformation is as follows:

1. **Add reference counts:** Add a reference count field `rc` to each object type to be used by the implementation. This field should be set to 1 in a newly-created object (in an object-oriented language such as C++, initialization may be achieved through object constructors).
2. **Provide an `LFRCDestroy(v)` function:** Write a function `LFRCDestroy(v)` that accepts a pointer `v` to an object. If `v` is `NULL`, then the function should simply return; otherwise it should atomically decrement `v->rc`. If the reference count field becomes zero as a result, `LFRCDestroy` should recursively call itself with each pointer in the object, and then free the object. An example is provided below, and we provide a function (`add_to_rc`) for the atomic decrement of the `rc` field, so writing this function is straightforward. We employ this function only because it is the most convenient and language-independent way to iterate over all pointers in an object. Other implementations may provide similar facility using language-specific constructs.



3. **Ensure no garbage cycles:** Ensure that the implementation does not result in referencing cycles in or among garbage objects. This step may be non-trivial or even impossible for some concurrent data structure implementations. If this property cannot be achieved for a particular data structure, then it is not a candidate for applying our methodology. Note that, as illustrated below, the concurrent data structure may include cycles. However, storage no longer reachable should not.
  
4. **Produce correctly-typed LFRC pointer operations:** We have provided code for the LFRC pointer operations to be used in the example implementation presented in the next section. In this implementation, there is only one type of pointer. For simplicity, we have explicitly designed the LFRC pointer operations for this type. For other simple concurrent shared object implementations, this step can be achieved by simply replacing the `SNode` type used in this implementation with a new type. In algorithms and data structure implementations that use multiple pointer types, a variety of alternative implementations are possible. In general, operations may be duplicated for the various pointer types or, alternatively, the code for the operations may be unified to accept different pointer types. For example, in some realizations an `rc` field may be defined uniformly, e.g., at the same offset in all objects, and `void` pointers may be employed instead of specifically-types ones. In such realizations, definition of multiple object-type-specific LFRC pointer operations can be eliminated. Nonetheless, for clarity of illustration, an object-type-specific set of LFRC pointer operations is described below.
  
5. **Replace pointer operations:** Replace each pointer operation with its LFRC pointer operation counterpart. For example, if `A0` and `A1` are pointers to shared pointer variables, and `x0`, `x1`, `old0`, `old1`, `new0`, `new1` are pointer variables, then replacements may be made as follows:

Replaced Pointer Operation	LFRC Pointer Operation
<code>x0 = *A0;</code>	<code>LFRCLoad (A0, &amp;x0);</code>
<code>*A0 = x0;</code>	<code>LFRCStore (A0, x0);</code>
<code>x0 = x1;</code>	<code>LFRCCopy (&amp;x0, x1);</code>
<code>CAS (A0, old0, new0)</code>	<code>LFRC CAS (A0, old0, new0)</code>
<code>DCAS (A0, A1, old0, old1, new0, new1)</code>	<code>LFRC CAS (A0, A1, old0, old1, new0, new1)</code>

Note that the table does not contain an entry for replacing an assignment of one shared pointer value to another, for example `*A0 = *A1`. Such assignments are not atomic. Instead, the location pointed to by A1 is read into a register in one instruction, and the contents of the register are stored into the location pointed to by A0 in a separate instruction. This approach should be reflected explicitly in a transformed implementation, e.g., with the following code:

```
{
    ObjectType *x = NULL;
    LFRCLoad(A1, &x);
    LFRCTore(A0, x);
    LFRCDestroy(x);
}
```

or its substantial equivalent, whether included directly or using a “wrapper” function.

6. **Management of local pointer variables:** Finally, whenever a thread loses a pointer (e.g., when a function that has local pointer variables returns, so its local variables go out of scope), it first calls `LFRCDestroy()` with this pointer. In addition, pointer variables are initialized to `NULL` before being used with any of the LFRC operations. Thus, pointers in a newly-allocated object should be initialized to `NULL` before the object is made visible to other threads. As illustrated in the example below, it is also important to explicitly remove pointers contained in a statically allocated object before destroying that object.

#### Example: Transformation of a Concurrent Deque Implementation

[1042] In this section, we show how to use our methodology to construct a GC-independent implementation of a concurrent double-ended queue (deque) object, based on a GC-dependent implementation presented in greater detail in U.S. Patent Application No. <not yet assigned> (docket 004-4666-1), entitled “LOCK-FREE IMPLEMENTATION OF CONCURRENT SHARED OBJECT WITH DYNAMIC NODE ALLOCATION AND DISTINGUISHING POINTER VALUE,” naming Guy L. Steele Jr., Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, Mark S. Moir and David L. Detlefs as inventors, and filed on even date herewith. The description, in the above-identified U.S. Patent Application, of a deque object implementation

Sub  
A2

(including supporting data structure representations and access operations) suitable for use in a garbage-collected storage environment is incorporated herein by reference in its entirety. Hereafter, a class of implementations patterned on such techniques is referred to variously as the "Snark" algorithm, data structure, access operations, implementation, etc.

[1043] Some features of a Snark algorithm, data structure, access operation and/or implementation are summarized in the context of **FIGS. 3A** and **3B** in preparation for explaining the application of the LFRC methodology thereto. In particular, C++ class declarations and code for one access operation, `pushRight` of a Snark implementation are shown in **FIGS. 3A** and **3B**. A corresponding transformed implementation employing LFRC pointer operations is shown in **FIGS. 4A** and **4B**. For clarity, uninteresting details such as access modifiers have been omitted.

[1044] Transformation of the `pushRight` access operation is illustrated in significant detail herein. Accordingly, based on the description herein, corresponding transformations of other Snark access operations, e.g., `pushLeft`, `popRight` and `popLeft` will be appreciated by persons of ordinary skill in the art. Indeed, a resultant set of access operations is described in the above-incorporated U.S. Patent Application. Furthermore, U.S. Patent Application No. <not yet assigned> (docket 004-4665-1), entitled "CONCURRENT SHARED OBJECT IMPLEMENTED USING A LINKED-LIST WITH AMORTIZED NODE ALLOCATION," naming Paul A. Martin, David L. Detlefs, Alexander T. Garthwaite, Guy L. Steele Jr. and Mark S. Moir as inventors, and filed on even date herewith, which is incorporated herein by reference, describes an explicit reclamation embodiment of another concurrent shared object implementation that was achieved by applying the methodology described herein.

[1045] In general, a Snark implementation represents a deque as a doubly-linked list. Nodes of the doubly-linked list are called `SNodes` (*see* lines 1-2, **FIG. 3A**). Nodes contain `L` and `R` pointers, which point to left and right neighbors, respectively. A Snark deque includes two "hat" pointers (`LeftHat` and `RightHat`), which respectively point to the leftmost and rightmost nodes in a non-empty deque, and a special "Dummy" node, which is used as a sentinel node at one or both ends of the

Sub  
A3

deque (*see* line 3, **FIG. 3A**). A Snark constructor (*see* lines 4-9, **FIG. 3A**) allocates an SNode for Dummy, makes the L and R pointers of Dummy point to Dummy itself, and makes LeftHat and RightHat both point to Dummy. This is one representation of an empty deque in this implementation.

[1046] Self-pointers in the L and R pointers of the Dummy node are used to distinguish this node from nodes that are currently part of the deque. Some pop operations leave a previous deque node as a sentinel node. In this case, a pointer in this node is also directed toward the node itself to facilitate identification of the node as a sentinel. Pointers in the Snark algorithm are accessed only by loading them, storing them, and applying a DCAS operation to them. Additional detail is provided in the above-incorporated U.S. Patent Application.

[1047] Corresponding aspects of GC-independent code derived from the original Snark algorithm by using the LFRC methodology are shown in **FIGS. 4A** and **4B**. **FIGS. 3A** and **4A** and **FIGS. 3B** and **4B** are presented in side-by-side fashion to illustrate the correspondence. Below, we briefly explain the differences between the two implementations and explain how we applied the LFRC methodology to the first implementation to derive the second. The explanation follows the transformation method outlined above.

1. We added a reference count field (*rc*) to the SNode object (*see* line 31, **FIG. 4A**). The SNode constructor sets the reference count to 1 (*see* line 32, **FIG. 4A**) to indicate that one pointer to the SNode exists (i.e., the pointer returned after allocation of storage by the constructor).
2. We wrote an LFRCDestroy function. One implementation of this function is shown below.

```
void LFRCDestroy(SNode *p) {
    if (p != Null && add_to_rc(p, -1)==1) {
        LFRCDestroy(p->L, p->R);
        delete p;
    }
}
```

where we use the notation `LFRCDestroy(a, b, ...)`, both here and elsewhere, as shorthand for calling or otherwise invoking `LFRCDestroy` for each argument.

3. The Snark algorithm required a slight modification in order to ensure that garbage does not include reference cycles. As explained above, sentinel nodes (either the Dummy node, or a deque node that becomes a sentinel node after being popped) include self-pointers. The correctness of the Snark algorithm is not affected by changing these self-pointers to NULL pointers, and interpreting NULL pointers the same way the original algorithm interprets self-pointers. Because NULL pointers are not employed the original algorithm (see **FIGS. 3A** and **3B**), each occurrence of a NULL pointer in the revised algorithm can be interpreted as a self-pointer. The result is that there are no cycles in garbage. To effect these changes, we changed the Snark constructor to set the Dummy node up with NULL pointers instead of self-pointers (see lines 36-37, **FIG. 4A**). In addition, we changed corresponding checks for self-pointers to checks for NULL pointers (see line 59, **FIG. 4B**). Finally, we changed the code in left and right pop operations (not shown) to install NULL pointers instead of self-pointers.
4. In this case, we did not have to do anything for Step 4, because the LFRC pointer operations (as presented in the next section) are already typed for Snark.
5. We replaced pointer accesses of the original algorithm (see **FIGS. 3A** and **3B**) with LFRC pointer operations (see lines 35-39, 42-44, **FIG. 4A** and lines 54, 57-58, 60-62 and 65-66, **FIG. 4B**), in accordance with the transformations shown above.
6. We inserted calls to `LFRCDestroy` for each time a pointer variable is destroyed such as before returns from a function that contain local pointer variables (see lines 52, 63 and 67, **FIG. 4B**). In the notation of **FIG. 4B**, a call to `LFRCDestroy` with multiple arguments is shorthand for calling

LFRCDestroy once with each argument. Also, we added initialization to NULL for all pointers (*see* lines 32 and 34, **FIG. 4A** and line 50, **FIG. 4B**). These initializations were not needed in the original algorithm, as the pointers are always modified before they are read. However, because LFRCStore must reduce the reference count of the object previously pointed to by the pointer it modifies, it is important to ensure that pointers are initialized before accessing them with a LFRC pointer operation. Finally, we added a destructor operation for Snark objects, which pops any remaining values and writes NULL to the LeftHat, RightHat, and Dummy pointers before destroying the Snark object itself (*see* lines 40-44, **FIG. 4A**). This ensures that objects remaining in the deque are freed. Note that since the illustrated implementation of the destructor omits synchronization, it is not intended to be invoked concurrently with other deque operations. Accordingly, the Snark object should be destroyed only after all threads have finished accessing it.

#### **Implementation of LFRC Pointer Operations**

[1048] In the description that follows, we describe an illustrative implementation of LFRC pointer operations and explain why the illustrative operations ensure that there are no memory leaks and that memory is not freed prematurely. The LFRC pointer operations maintain a reference count in each object, which reflects the number of pointers to the object. When this count reaches zero, there are no more pointers to the object and the object can be freed.

[1049] The main difficulty is that we cannot atomically change a pointer variable from pointing to one object to pointing to another and update the reference counts of both objects. We overcome this problem with the observations that:

1. provided an object's reference counter is always *at least* the number of pointers to the object, it will never be freed prematurely, and
2. provided the count becomes zero when there are no longer any pointers to the object, there are no memory leaks.

[1050] Thus, we conservatively increment an object's reference count *before* creating a new pointer to it. If we subsequently fail to create that pointer, then we can decrement the reference count again afterwards to reflect that the new pointer was not created. An important mechanism in the illustrated implementation is the use of DCAS to increment an object's reference count while simultaneously checking that some pointer to the object exists. This avoids the possibility of updating an object after it has been freed, thereby potentially corrupting data in the heap, or in an object that has been reallocated.

[1051] We now describe a lock-free implementation of the LFRCLoad pointer operations, beginning with an implementation of LFRCLoad as follows:

```

1 void LFRCLoad(SNode **A, SNode **dest) {
2     SNode *a, *olddest = *dest;
3     long r;
4     while (true) {
5         a = *A;
6         if (a == Null) {
7             *dest = Null;
8             break;
9         }
10        r = a->rc;
11        if (DCAS(A, &a->rc, a, r, a, r+1)) {
12            *dest = a;
13            break;
14        }
15    }
16    LFRCDestroy(olddest);
17 }
```

[1052] where LFRCLoad accepts two parameters, a pointer A to a shared pointer, and a pointer dest to a local pointer variable of the calling thread. The semantics of the LFRCLoad operation is to load the value in the location pointed to by A into the variable pointed to by dest. This has the effect of destroying one pointer (the previous value in the location pointed to by dest) and creating another (the new value of \*dest). Thus, we must potentially update two reference counts. The LFRCLoad operation begins by recording the previous value of the pointer (line 2), so that it can be destroyed later. Note that we cannot destroy it yet, as this would risk destroying the object to which we are about to create a pointer.

[1053] Next, the LFRCLoad operation loads a new value from \*A and, if the pointer read is non-NULL, increments the reference count of the object pointed to by \*A in order to record that a new pointer to this object has been created. In this case, because the calling thread does not (necessarily) already have a pointer to this object, it is not safe to update the reference count using a simple CAS primitive. The object might be freed before the CAS executes, creating a risk that execution of the CAS modifies a location in a freed object or in an object that has subsequently been reallocated for another purpose. Therefore, the LFRCLoad operation uses a DCAS primitive to attempt to atomically increment the reference count, while ensuring that the pointer to the object still exists.

[1054] In the above implementation, these goals are achieved as follows. First, the LFRCLoad operation reads the contents of \*A (line 5). If it sees a NULL pointer, there is no reference count to be incremented, so LFRCLoad simply sets \*dest to NULL (lines 6-8). Otherwise, it reads the current reference count of the object pointed to by the pointer it read in line 5, and then attempts to increment this count using a DCAS (line 11) to ensure that the pointer to the object containing the reference count still exists. Note that there is no risk that the object containing the pointer being read by LFRCLoad is freed during the execution of LFRCLoad because the calling thread has a pointer to this object that is not destroyed during the execution of LFRCLoad. Accordingly, the reference count cannot fall to zero. If the DCAS succeeds, then the value read is stored (line 12) in the variable passed to LFRCLoad for this purpose. Otherwise, LFRCLoad retries. After LFRCLoad succeeds in either loading a NULL pointer, or loading a non-NULL pointer and incrementing the reference count of the object to which it points, it calls LFRCDestroy in order to record that the pointer previously in \*dest has been destroyed (line 16).

[1055] An illustrative implementation of the LFRCDestroy operation will be understood as follows:

```

1 void LFRCDestroy(SNode *p) {
2     if (p != Null && add_to_rc(p, -1)==1) {
3         LFRCDestroy(p->L, p->R);
4         delete p;
5     }
6 }

```



[1056] If the LFRCDestroy operation's argument is non-NULL, then it decrements the reference count of the object pointed to by its argument (line 2, above). This is done using an add\_to\_rc function (such as that shown below) implemented using a CAS primitive. The add\_to\_rc function is safe (in the sense that there is no risk that it will modify a freed object) because it is called only in situations in which we know that the calling thread has a pointer to this object, which has previously been included in the reference count. Therefore, there is no risk that the reference count will become zero, thereby causing the object to be freed, before the add\_to\_rc function completes. If execution of the add\_to\_rc function causes the reference count to become zero, then we are destroying the last pointer to this object, so it can be freed (line 4, above). First, however, LFRCDestroy calls itself recursively (line 3, above) with each pointer in the object in order to update the reference counts of objects to which the soon-to-be-freed object has pointers.

```

1  long add_to_rc(SNode *p,  int v) {
2      long oldrc;
3      while (true) {
4          oldrc = p->rc;
5          if (CAS(&p->rc), oldrc, oldrc+v)
6              return oldrc;
7      }
8  }
```

[1057] An LFRCStore operation can be implemented as follows:

```

1  void LFRCStore(SNode **A, SNode *v) {
2      SNode *oldval;
3      if (v != Null)
4          add_to_rc(v, 1);
5      while (true) {
6          oldval = *A;
7          if (CAS(A, oldval, v)) {
8              LFRCDestroy(oldval);
9              return;
10         }
11     }
12 }
```

[1058] where the LFRCStore operation accepts two parameters, a pointer A to a location that contains a pointer, and a pointer value v to be stored in this location. If the value v is not NULL, then the LFRCStore operation increments the reference count of the object to which v points (lines 3-4). Note that at this point, the new

pointer to this object has not been created, so the reference count is greater than the number of pointers to the object. However, this situation will not persist past the end of the execution of the LFRCTestore operation, since LFRCTestore does not return until that pointer has been created. In the illustrated implementation, the pointer is created by repeatedly reading the current value of the pointer and using a CAS primitive to attempt to change the contents of the location referenced by A to the pointer value v (lines 5-9). When the CAS succeeds, we have created the pointer previously counted and we have also destroyed a pointer, namely the previous contents of \*A. Therefore, LFRCTestore calls LFRCTestroy (line 8) to decrement the reference count of the object to which the now-destroyed pointer points.

[1059] Finally, a LFRCTestCAS operation can be implemented as follows:

```

1  bool LFRCTestCAS(SNode **A0,  SNode **A1,
2                    SNode *old0,  SNode *old1,
3                    SNode *new0,  SNode *new1) {
4      if (new0 != Null) add_to_rc(new0, 1);
5      if (new1 != Null) add_to_rc(new1, 1);
6      if (DCAS(A0, A1, old0, old1, new0, new1)) {
7          LFRCTestroy(old0, old1);
8          return true;
9      } else {
10         LFRCTestroy(new0, new1);
11         return false;
12     }
13 }
```

[1060] where the LFRCTestCAS operation accepts six parameters, corresponding to the DCAS parameters described earlier. The illustrated implementation of the LFRCTestCAS operation is similar to that of the LFRCTestore operation in that it increments the reference counts of objects before creating new pointers to them (lines 4-5) using the add\_to\_rc function, thereby temporarily setting these counts artificially high. However, the LFRCTestCAS operation differs from the LFRCTestore operation in that it does not insist on eventually creating those new pointers. If the DCAS at line 6 fails, then LFRCTestCAS calls LFRCTestroy for each of the objects whose reference counts were previously incremented, thereby compensating for the previous increments and then returning false (see lines 9-11). On the other hand, if the DCAS succeeds, then the previous increments were justified but we have destroyed two pointers, namely the previous values of the two locations updated by the DCAS. Therefore, the

LFRCDCAS operation calls LFRCDestroy to decrement the reference counts of (and potentially free) the corresponding objects and then returns true (*see* lines 7-8). One suitable implementation of an LFRCCAS operation (not shown) is just a simplification of the LFRCDCAS with handling of the second location omitted.

[1061] While the above-described LFRCLoad, LFRCStore, LFRCDCAS and LFRCDestroy operations are sufficient for the illustrated implementation (or transformation) of a pushRight access operation for a concurrent shared object implementation of a deque, other access operations, other implementations and other concurrent shared objects may employ alternative or additional LFRC pointer operations. For example, in some implementations, an LFRCCopy operation may be useful. One implementation is as follows:

```

1 void LFRCCopy(SNode **v, SNode *w) {
2     if (w != Null)
3         add_to_rc(w,1);
4     LFRCDestroy(*v);
5     *v = w;
6 }

```

[1062] where the LFRCCopy operation accepts two parameters, a pointer *v* to a local pointer variable, and a value *w* of a local pointer variable. The semantics of this operation is to assign the value *w* to the variable pointed to by *v*. This creates a new pointer to the object referenced by *w* (if *w* is not NULL), so LFRCCopy increments the reference count of that object (lines 2-3). The LFRCCopy operation also destroys a pointer, namely the previous contents of *\*v*, so LFRCCopy calls LFRCDestroy (line 4) to decrement the reference count of the object referenced by the now-destroyed pointer. Finally, LFRCCopy assigns the value *w* to the pointer variable pointed to by *v* and returns.

[1063] Other LFRC operations that may be useful in some implementations include a variant of the previously described LFRCLoad operation suitable for use in situations where the target of the load cannot contain a pointer. For example, such a variation may be implemented as follows:

```

1 void LFRCLoad(SNode **A) {
2     SNode *a;
3     long r;

```

```

4   while (true) {
5       a = *A;
6       if (a == NULL)
7           return NULL;
8       r = a->rc;
9       if (DCAS(A,&a->rc,a,r,a,r+1))
10          return a;
11     }
12 }

```

[1064] Other suitable LFRC operations include the previously-described LFRCStoreAlloc operation, which may be implemented as follows:

```

1   void LFRCStoreAlloc(SNode **A, SNode *v) {
2       SNode *oldval;
3       while (true) {
4           oldval = *A;
5           if (CAS(A,oldval,v)) {
6               LFRCDestroy (oldval);
7               return;
8           }
9       }
10  }

```

[1065] in situations in which we want to invoke an allocation routine directly as the second parameter of a LFRC store operation. In addition, some implementations or transformations may include a variant of the LFRCDCAS operation such as the following:

```

1   bool LFRCDCAS1(SNode **a0, int *a1, SNode *old0, int old1,
2   SNode *new0, int new1){
3
4       if (new0 != NULL)
5           add_to_rc(new0,1);
6
7       if (DCAS(a0,a1,old0,old1,new0,new1)) {    // Do DCAS
8           LFRCDestroy(old0);
9           return true;
10      } else {
11          LFRCDestroy(new0);
12          return false;
13      }
14  }

```

[1066] where the second location operated upon by the DCAS pointer operation contains a literal (e.g., an integer) rather than a pointer.

[1067] Some implementations or transformations may exploit other LFRC pointer operations such as the previously described LFRCPass operation, which may be implemented as follows:

```

1  SNode* LFRCPass(SNode *p) {
2      if (p!=NULL)
3          add_to_rc(p,1);
4      return p;
5  }

```

[1068] where the LFRCPass function may be employed to facilitate passing a pointer by value while appropriately maintaining a corresponding reference count. These and other variations on the illustrated set of LFRC pointer operations will be appreciated by persons of ordinary skill in the art based on the description herein.

[1069] While the invention has been described with reference to various embodiments, it will be understood that these embodiments are illustrative and that the scope of the invention is not limited to them. Terms such as always, never, all, none, etc. are used herein to describe sets of consistent states presented by a given computational system. Of course, persons of ordinary skill in the art will recognize that certain transitory states may and do exist in physical implementations even if not presented by the computational system. Accordingly, such terms and invariants will be understood in the context of consistent states presented by a given computational system rather than as a requirement for precisely simultaneous effect of multiple state changes. This “hiding” of internal states is commonly referred to by calling the composite operation “atomic”, and by allusion to a prohibition against any process seeing any of the internal states partially performed.

[1070] Many variations, modifications, additions, and improvements are possible. For example, while application to a particular concurrent shared object (i.e., a deque) and a particular implementation thereof (i.e., a Snark realization) has been described in detail herein, applications to other shared objects and other implementations will also be appreciated by persons of ordinary skill in the art. In addition, more complex shared object structures may be defined that exploit the techniques described herein. Other synchronization primitives may be employed. Plural instances may be provided for components, operations or structures described herein as a single instance. Finally, boundaries between various components, operations and data stores are somewhat arbitrary, and particular operations are illustrated in the context of specific illustrative configurations. Other allocations of functionality are envisioned and may fall within the scope of claims that follow. In particular, allocations of pointer

operation functionality to particular procedures, functions or methods are merely illustrative and other implementations may provide similar functionality in-line, using a different allocation of functionality to procedures, functions or methods, or through use of overloaded, object-specific pointer operations, smart pointers or other similar facility.

[1071] In general, structures and functionality presented as separate components in the exemplary configurations may be implemented as a combined structure or component. Similarly, structures and functionality presented as a single component may be implemented as separate components. These and other variations, modifications, additions, and improvements may fall within the scope of the invention as defined in the claims that follow.